UNITÉ DE RECHERCHE
INRIA-RENNES

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél.:(1) 39 63 55 11

Rapports de Recherche

N° 1489

*Programme 3*
*Intelligence artificielle, Systèmes cognitifs et*
*Interaction homme-machine*

# PROGRAMMATION D'UN NOYAU UNIX EN GAMMA

**Pascale LE CERTEN**
**Hector RUIZ BARRADAS**

**Juillet 1991**

*RR-1489*

IRISA

INSTITUT DE RECHERCHE EN INFORMATIQUE
ET SYSTÈMES ALÉATOIRES

Campus Universitaire de Beaulieu
35042 - RENNES CÉDEX
FRANCE
Téléphone : 99 36 20 00
Télex : UNIRISA 950 473 F
Télécopie : 99 38 38 32

# Programmation d'un noyau UNIX en GAMMA

**Publication Interne n° 594 - Juillet 1991, 48 pages**
**Programme 3**

Pascale Le Certen
IRISA/Université de Rennes I
Campus de Beaulieu
35042 Rennes Cedex. FRANCE.
*e-mail : lecerten@irisa.fr*

Héctor Ruíz Barradas*
IRISA/INRIA
Campus de Beaulieu
35042 Rennes Cedex. FRANCE.
*e-mail : ruiz@irisa.fr*

*Boursier du XII$^{ème}$ programme Franco-Mexicain CONACyT-CEFI

CENTRE NATIONAL DE LA RECHERCHE SCIENTIFIQUE
(L. A. 227)
UNIVERSITÉ DE RENNES 1 I. N. S. A. DE RENNES

INSTITUT NATIONAL DE RECHERCHE
EN INFORMATIQUE ET EN AUTOMATIQUE
(LABORATOIRE DE RENNES)

## Programmation d'un noyau UNIX en GAMMA

Résumé

Dans ce document, nous montrons qu'il est possible de programmer des systèmes opératoires à l'aide d'une combinaison du formalisme GAMMA et d'un langage de programmation fonctionnelle (GAML). Notre démarche consiste à décrire, à l'aide de GAML, un noyau de système UNIX. Nous donnons un aperçu de la mise en œuvre de la gestion des processus, de la gestion des fichiers et de l'interpréteur de commandes. En conclusion, nous exhibons les avantages généraux qu'offrent le formalisme GAMMA et la programmation fonctionnelle pour la spécification de systèmes.

## Programming a UNIX kernel in GAMMA

Abstract

In this paper, we show that programming operating systems is possible using a combination of GAMMA formalism and functional programming (GAML). Our approach consists in describing the design of a UNIX system kernel using GAML. We present a part of the implementation of process management, file system and command interpreter. In conclusion, we display the general advantages of the Γ formalism and functional programming for systems specifications.

# Table des matières

# 1 Introduction

## 1.1 Contexte

Actuellement, une voie de recherche importante dans le domaine du parallélisme consiste à étudier les possibilités offertes par les langages fonctionnels pour l'expression de programmes s'exécutant de façon concurrente. Etant donné l'intérêt croissant pour ces langages, il semble naturel d'étudier leur adaptation à des domaines particuliers. Entre autres, les langages fonctionnels ne semblent pas appropriés à la programmation d'une certaine classe de problèmes. Les programmes interactifs et les programmes accédant des ressources externes en font partie. Les systèmes d'exploitation constituent un exemple important de ce type de programmes.
La plupart des travaux abordant ce sujet sont basés sur l'étude de réseaux de processus communicants. Ces réseaux exploitent l'évaluation paresseuse des programmes fonctionnels pour mettre en œuvre des programmes interactifs. Les processus sont des fonctions admettant des flots comme argument d'entrée et produisant des données dans un flot de sortie. Citons pour exemples les travaux de S.B. Jones et A.F. Sinclair [3], W. Stoye [6] et D. Turner [8]. Un des principaux problèmes que posent leur solution est l'utilisation explicite ou implicite d'une pseudo-fonction *fusionneur* (merge). La nécessité de son utilisation est claire : elle sert à résoudre les problèmes de non-déterminisme.
C'est cette raison qui nous pousse à étudier les possibilités offertes par le formalisme GAMMA ($\Gamma$), défini à l'IRISA, pour exprimer des systèmes d'exploitation. Ce formalisme apporte en effet une solution immédiate au problème de non-déterminisme.
Notre travail va consister à montrer la faisabilité de la programmation de système en utilisant une combinaison du formalisme $\Gamma$ et de la programmation fonctionnelle. Plus précisément, nous nous proposons de décrire à l'aide de cette combinaison, un noyau de système d'exploitation, celui du système UNIX [1].

## 1.2 Contenu du document

Ce document décrit la programmation des constituants principaux du système.
Dans la première partie, nous présentons brièvement le formalisme GAMMA et donnons quelques indications concernant le formalisme obtenu en combinant $\Gamma$ et le langage fonctionnel CAML.
La deuxième partie décrit la mise en œuvre du système de gestion des processus. Nous exposons par exemple, comment s'effectuent la création de processus (fork), la communication entre processus.
Dans la troisième partie, nous donnons un aperçu de la programmation du système de gestion de fichiers. En particulier, nous décrivons les services offerts, l'architecture du système et nous exhibons des programmes GAMMA correspondant au traitement de requêtes spécifiques.

[1] UNIX est une marque déposée des laboratoires BELL

Nous donnons, dans la quatrième partie, un programmation possible de l'interpréteur de commandes (shell) et montrons comment peuvent être spécifiées certaines commandes et utilitaires.
Dans une dernière partie, nous analysons les résultats et exhibons les avantages des outils utilisés et de la programmation obtenue.
En conclusion, nous précisons l'état d'avancement de nos travaux et évoquons quelques perspectives.

# 2 Formalisme GAMMA

GAMMA est un outil de spécification d'algorithmes basé sur un principe de transformation d'un ensemble d'objets (multiensemble) sur lui-même. Nous exposons ici le principe de fonctionnement d'un programme GAMMA et illustrons notre propos à l'aide de quelques exemples.

## 2.1 Présentation générale

La structure de donnée de base est le multiensemble qui est un ensemble pouvant contenir plusieurs occurrences d'un même élément. Les composants atomiques des multiensembles peuvent être de type réel, caractère, entier, tuple ou multiensemble. Une spécification Gamma $\Gamma(R, A)$consiste en l'énoncé de:

- une condition de réaction, $R(X_1, \ldots, X_n)$, expression booléenne
- une action $A(X_1, \ldots, X_n)$, expression de type multiensemble

La signification est la suivante:

L'état initial est un multiensemble $M_0$; tant que la condition de réaction est applicable, c'est à dire tant qu'il existe $\{X_1, \ldots, X_n\}$ qui satisfait $R$, un tel sous-multiensemble est choisi de façon indéterministe et l'action $A$ est exécutée : $\{X_1, \ldots, X_n\}$ est remplacé par le résultat de l'application $A(X_1, \ldots, A_n)$:
Autrement dit,

$$\begin{aligned} \Gamma(R, A)M \ :* \ & [ \\ & \exists \{X_1, ..., X_n\} \in M, R(X_1, ..., X_n) \rightarrow \\ & M := M - \{X_1, ..., Xn\} + A(X_1, ..., X_n) \\ & ] \end{aligned}$$

Cette définition s'étend naturellement à plusieurs couples de fonctions (Ri,Ai). Le calcul consiste donc en une succession d'applications de règles qui consomment des éléments du multiensemble tout en en produisant de nouveaux. Un élément ne peut participer simultanément à plusieurs réactions, mais plusieurs réactions peuvent s'exécuter simultanément sur des ensembles disjoints d'éléments.Grâce à cette propriété, il est souvent possible de donner une interprétation très parallèle aux programmes GAMMA.

Divers travaux sont en cours, visant l'étude des possibilités du formalisme GAMMA. En particulier, afin de rendre plus aisée l'écriture d'algorithmes, une syntaxe adaptée à l'opérateur $\Gamma$ a été définie, qui permet son intégration au langage CAML. C'est ce nouveau langage appelé GAML que nous utilisons dans la suite de ce document, pour décrire des algorithmes.

## 2.2 Exemples de programmes GAML

Afin de donner un aperçu de la structure des programmes GAML nous présentons quatre exemples.

**Exemple 1**

```
let elimine_doubles =
   (gam M -> M
      rp (x,y): M
      if  x = y
      by  x: M
   )
```

Ceci définit un programme GAMMA identifié par *elimine_doubles* qui prend en argument un multiensemble $M$ et rend en résultat ce multiensemble $M$, éventuellement modifié. Ce programme est décrit par une seule réaction qui indique que deux éléments $x,y$ de $M$ vont réagir lorsque $x$ et $y$ ont des valeurs égales. Ces deux éléments sont ôtés de $M$ et remplacés par la valeur de $x$. En d'autres termes, la réaction signifie tout simplement: "remplacer $(x,y) \subseteq M$ par $x$ si $x = y$. L'exécution du programme *elimine_doubles* se termime lorsque tous les éléments de $M$ ont des valeurs distinctes.

**Exemple 2**

Il est possible de définir un programme GAMMA travaillant sur plusieurs multiensembles. Ainsi par exemple:

```
(gam E,F -> E,G
   rp x:E,y:F
   if x = y
   by x:G,y:F
)
```

est un programme qui admet en argument le couple *(E,F)* où $E$ et $F$ sont de type multiensemble, et rend en résultat la couple de multiensembles *(E,G)* (avant l'exécution du programme, $G$ est initilalisé à $\emptyset$). Le nombre de multiensembles passés en argument ou rendus en résultat n'est pas limité.

**Exemple 3**

Un programme qui ôte les éléments d'un multiensemble sous certaines conditions peut s'exprimer comme suit:

```
(gam M -> M
   rm x:M
   if P(x)
)
```

*rm* se lit "remove". Ce programme enlève les éléments $x$ de $M$ qui satisfont la propriété *P(x)*.

**Exemple 4**

Afin de ne pas alourdir une condition de réaction, nous utilisons souvent l'appariement de formes (*pattern matching*); ainsi par exemple:

```
rp (x,y), (x,z):M
by (y,z):M
```

est équivalent à:

```
rp (x,y),(x',z):M
if  x = x'
by (y,z):M
```

# 3 Cœur du système : gestion des processus

## 3.1 Introduction

Dans un système d'exploitation, les problèmes les plus importants résident dans l'organisation et la gestion des processus. C'est pourquoi nous avons choisi de présenter en premier lieu une solution GAMMA à ces problèmes.
Le programme GAMMA mettant en œuvre la gestion de processus *type unix* a le profil:

```
P_SYS : (SYS,BAL,BLK,PIDS,PROC) -> (SYS,BAL,BLK,PIDS,PROC)
```

Il prend donc cinq arguments de type multiensemble. Le contenu de chaque multiensemble est le suivant:

- *SYS* contient les processus prêts à s'exécuter.
  Un processus est représenté par un triplet *(P,arg,pid)* où
  - $P$ est une expression composée de fonctions et de programmes GAMMA mettant en œuvre le processsus.
  - *arg* est une liste contenant les arguments de l'expression $P$. Celle-ci a la forme $\langle M_1, \ldots, M_n \rangle$ où chaque $M_i$ est un multiensemble contenant des données nécessaires au processus. La longueur d'*arg* est au moins égale à un. Le premier élément de cette liste ($M_1$) représente la boîte aux lettres associée au processus. Un message transmis par l'intermédiaire de cette boîte aux lettres est représenté par un n_uplet dont la structure dépend de la nature du message.
  - *pid* est l'identificateur de processus.
- *BAL* contient les messages adressés au noyau $P_{sys}$.
- *BLK* contient les processus bloqués. Ceux-ci sont les processus qui, au moyen de la requête *wait*, ont demandé l'attente de la terminaison d'un processus fils.
- *PIDS* contient les identificateurs libres de processus.
- *PROC* contient les informations sur les dépendances entre les processus. Ces caractéristiques sont représentées par des quadruplets de la forme *(pid, pid_père,l_fils,wait)* où
  - *pid* est l'identificateur d'un processus,
  - *pid_père* est l'identificateur du processus père,
  - *l_fils* est la liste contenant les identificateurs des processus fils,
  - *wait* est un champ booléen indiquant si le processus est en attente de la terminaison d'un de ses fils.

## 3.2 Services offerts

Les services offerts par le noyau *P_SYS* sont les suivants:

- exécution de processus,
- communication (*send*),
- création de processus (*fork*),
- attente de processus (*wait*),
- terminaison de processus (*exit*),

- remplacement de code (*exec*).

Mis à part l'exécution de processus, ces services nécessitent, pour être exécutés, la présence dans la boîte aux lettres de *P_SYS* de requêtes associées.

### 3.2.1 Exécution de processus

La réaction du programme *P_SYS* mettant en œuvre l'exécution de processus est:

```
rp (P,arg,pid):SYS
by (P,arg'',pid):SYS, req:BAL
    where
    bal = head(arg)
    arg' = P(arg)
    bal' = head(arg')
    req = bal' - bal
    bal''= bal ∩ bal'
    arg''= cons(bal'',tail(arg'))
```

Comme indiqué au paragraphe 3.1, un processus est représenté par un triplet de la forme *(P,arg,pid)*. Son exécution correspond à l'application de l'expression $P$ à l'argument *arg*.

L'application *P(arg)* rend en résultat *arg'*. Cette liste contient le même nombre de multiensembles que *arg*, mais ceux-ci peuvent avoir été modifiés. Après l'exécution de $P$, sa boîte aux lettres peut contenir deux sortes de messages: les messages adressés au processus $P$ qui n'ont pas été traités et les messages demandant les services de *P_SYS*. Ces derniers sont ôtés de la boîte aux lettres de $P$ et placés dans la boîte aux lettres du noyau *P_SYS* afin que les services demandés soient exécutés. Pour assurer le fonctionnement correct du programme *P_SYS*, il faut évidemment supposer l'existence d'un choix équitable, ceci afin qu'un processus en mèsure d'être exécuté le soit au bout d'un temps fini.

### 3.2.2 Communication entre processus

Le service de communication du noyau est utilisé pour le tranfert de messages entre les processus du système. Les réactions qui mettent en œuvre ce service sont:

```
rp (send,pid,msg):BAL, (P,arg,pid):SYS
by (P,arg',pid):SYS
    where
    arg' = cons(bal',tail(arg))
    bal' = head(arg) ∪ {msg}

rp (send,sys,msg):BAL
by  msg:BAL
```

Le n_uplet *(send,pid,msg)* représente la demande d'envoi du message *msg* à un processus dont l'identificateur est *pid*. Un tel élément présent dans *BAL* réagit avec le triplet de *SYS* représentant *pid*. L'action associée traduit le dépôt de *msg* dans la boîte aux lettres du processus destinataire.

La deuxième réaction traite le cas particulier des messages adressés au noyau *P_SYS*. Elle a pour effet de placer le message dans la boîte aux lettres du noyau.

### 3.2.3 Création de processus

La création de processus est faite par la primitive *(fork,pid)* où *pid* est l'identificateur du processus père qui demande la création. Cette primitive est mise en œuvre par la réaction suivante:

```
rp (fork,pid):BAL, (P,arg,pid):SYS, pid':PID,
   (pid,pid_père,l_fils,wait):PROC
by (P,arg',pid),(P,arg'',pid'):SYS,(send,FS,msg):BAL
   (pid,pid_père,l_fils',wait):PROC,(pid',pid,l_fils'',wait'):PROC
    where
    arg' = cons(bal',tail(arg))
    bal' = head(arg) ∪ {msg1}
    msg1 = (done_fork,pid')
    arg'' = cons(bal'',tail(arg))
    bal'' = {msg2} ∪ {msg3}
    msg2 = (done_fork,0)
    msg3 = (pid_fils,pid')
    msg = (do_fork,sys,(pid,pid'))
    l_fils' = cons(pid',l_fils)
    l_fils''= NIL
    wait' = FALSE
```

Ce programme indique une réaction possible entre les éléments suivants:

- le message *(fork,pid)*,
- le triplet *(P,arg,pid)* représentant le processus père qui demande la création de processus,
- les caractéristiques de ce même processus *(pid,pid_père, l_fils,wait)* ,
- l'identificateur du processus fils *pid'*.

L'exécution de la primitive *fork* entraîne la création du processus fils représenté par *(P,arg",pid')*. Les expressions mettant en œuvre les processus père et fils sont les mêmes. Les messages d'acquittement *msg1* et *msg2* sont utilisés pour distinguer dans *P* les codes mettant en œuvre le processus père ou fils. Le processus père reçoit le message *(done_fork,pid')* où

*pid'* est l'identificateur du processus fils. Le processus fils reçoit le message *(done_fork,0)*. Le message *msg1* est stocké dans la boîte aux lettres de l'argument *arg'* et le message *msg2* dans la boîte aux lettres de l'argument *arg"*. Le message *msg3* est utilisé pour faire connaître au processus fils son propre identificateur de processus.

L'identificateur *pid'* est ajouté dans la liste *l_fils* représentant les fils du processus père. Le triplet *(pid',pid,l_fils",wait')* dans *PROC* indique les caractéristiques du processus fils.

Les fichiers ouverts par le processus père doivent également apparaître dans le contexte du processus fils. Pour ce faire, le message *(do_fork,sys,(pid,pid'))* est envoyé au processus gestionnaire de fichiers.

### 3.2.4 Attente de processus

Un processus père peut demander l'attente de la fin d'exécution d'un processus fils au moyen de la requête *wait*. Cette requête est mise en œuvre par:

```
rp (wait,pid):BAL, (P,arg,pid):SYS, (pid,pid_père,l_fils,wait):PROC
by (P,arg,pid):BLK, (pid,pid_père,l_fils,wait'):PROC
    where
    wait' = TRUE
```

Le processus père est placé dans le multiensemble *BLK* (contenant des processus en attente d'un événement). *wait'* indique que l'exécution du processus est suspendue jusqu'à ce qu'un de ses processus fils termine son exécution.

### 3.2.5 Terminaison de processus

Pour signaler au noyau la fin de son exécution, un processus utilise la requête *(exit,pid,stat)* où *stat* est un code indiquant son état d'exécution. Les réactions correspondantes à cette requête sont les suivantes:

```
rp (exit,pid,stat):BAL, (P,arg,pid):SYS, (pid,pid_père,l_fils,wait),
   (pid_père,pid_ancêtre,l_fils',wait'):PROC
if (wait' = FALSE) and (l_fils = NIL)
by (pid_père,pid_ancêtre,l_fils'',wait'):PROC, (send,FS,msg):BAL
    where
    msg = (do_exit,sys,pid)
    l_fils'' = delete(l_fils',pid)

rp (exit,pid,stat):BAL, (P,arg,pid):SYS, (P',arg',pid_père):BLK
   (pid,pid_père,l_fils,wait), (pid_père,pid_ancêtre,l_fils',wait'):PROC
if (wait' = TRUE) and (l_fils = NIL)
by (P',arg'',pid_père):SYS, (pid_père,pid_ancêtre,l_fils'',wait''):PROC
   (send,FS,msg):BAL
    where
```

```
    msg = (do_exit,sys,pid)
    arg'' = cons(bal'',tail(arg'))
    bal'' = head(arg') ∪ {msg}
    msg = (done_exit,pid,stat)
    l_fils''= delete(l_fils',pid)
    wait'' = FALSE

rp (exit,pid,stat):BAL,(pid,pid_père,l_fils,wait),
   (pid_fils,pid,l_fils',wait'):PROC
if (head(l_fils) = pid_fils)
by (exit,pid,stat):BAL,(pid,pid_père,l_fils'',wait),
   (pid_fils,INIT,l_fils',wait'):PROC
    where
    l_fils'' = tail(l_fils)
```

La mise en œuvre de cette requête nécessite trois réactions car il faut distinguer les trois situations suivantes:

- le processus qui termine n'a pas de fils et le processus père n'attend pas,
- le processus qui termine n'a pas de fils,
- le processus qui termine a au moins un fils.

Dans le premier cas, il faut effectuer les actions suivantes:

- enlever de *SYS* la représentation du processus qui vient de terminer,
- enlever de *PROC* les caractéristiques de ce processus,
- modifier la liste *l_fils'* du processus père. L'identificateur du processus venant de terminer est effacé de la liste *l_fils*.

Dans le cas où le processus père attend la terminaison d'un de ses fils, il faut de plus:

- faire passer l'état du processus père de "bloqué" à "prêt à exécution". Pour ce faire, le triplet représentant le processus père est retiré de *BLK* et déposé dans *SYS*. Le champ *wait'* des caractéristiques du processus père est mis à *FALSE*.
- envoyer le message *msg* au processus père. Ce message contient l'identificateur du processus venant de terminer et le code indiquant l'état de son exécution.

Dans le cas où le processus qui termine possède des fils, il est nécessaire de modifier la donnée *pid_père* de tous ses fils : le champ indiquant le processus père de chaque fils est remplacé par *INIT*, qui est l'identificateur du créateur de tous les processus. Chaque fois que les caractéristiques d'un processus fils sont changées, l'identificateur de ce processus est effacé de la liste *l_fils'*. Cette réaction est répétée jusqu'à ce que toutes les caractéristiques des processus fils aient été modifiées. Après ces modifications, la requête de terminaison peut être traitée par la réaction appropriée.

### 3.2.6 Remplacement de code

Un processus peut utiliser la requête *exec* pour remplacer son image code. Pour ce faire le processus doit fournir le nom d'un fichier contenant le nouveau code. La mise en œuvre de cette requête implique la lecture du fichier et l'affectation de son contenu au processus demandeur de la requête. Elle est réalisée par les réactions suivantes:

```
rp (exec,nom_fich,argv,pid):BAL, (P,arg,pid):SYS
by (send,FS,msg1),(send,FS,msg2):BAL, (P,arg,pid,nom_fich,argv):BLK
    where
    msg1 = (open,sys,nom_fich)
    msg2 = (fstat,sys,nom_fich)

rp (done_open,FS,(nom_fich,cid)),(done_fstat,FS,(nom_fich,stat)):BAL
   (P,arg,pid,nom_fich,argv):BLK
if  cid ≠ NoSuchFile
by (send,FS,msg):BAL, (P,arg,pid,cid,argv):BLK
    where
    msg = (read,sys,(cid,stat.length))

rp (done_open,FS,(nom_fich,cid)),(done_fstat,FS,(nom_fich,stat)):BAL
   (P,arg,pid,nom_fich,argv):BLK
if (cid = NoSuchFile) and (stat = NoSuchFile)
by (P,arg',pid):SYS
    where
    arg' = cons(bal',tail(arg))
    bal' = head(arg) ∪ {msg}
    msg = (done_exec,sys,nom_fich,argv,NoSuchFile)

rp (done_read,FS,(cid,data)):BAL, (P,arg,pid,cid,argv):BLK
by (P',arg',pid):SYS
    where
    P' = F(data)
    arg' = argv
```

La première réaction est utilisée pour demander l'ouverture du fichier et la lecture des informations concernant le fichier. Le n_uplet représentant le processus est modifié à l'aide des arguments de l'appel et placé dans *BLK*. Cette modification est faite car il faut sauvegarder les arguments de l'appel jusqu'à ce que les données du fichier soient lues.

Si la réponse du processus gérant le système de fichiers indique que le fichier n'existe pas (*cid* = *NoSuchFile*), il faut alors relancer le processus demandeur et signaler l'échec. Pour ce faire le triplet représentant le processus est replacé dans *SYS* et sa boîte aux lettres contient un message indiquant le problème rencontré. Si le fichier existe, il faut demander la lecture du fichier. La longueur du fichier à lire est obtenue à partir des données reçues (*stat.length*).

Lors de la réception des données du fichier dans la dernière réaction, le code du processus *P* est remplacé. Le triplet *(P',arg',pid)* représente le processus modifié. *P'* est obtenu par l'application de la fonction *F* aux données du fichier reçu. Cette fonction est utilisée pour

coder les données du fichier d'une manière adéquate. L'argument *argv* doit être une liste de multiensembles.

# 4 Système de gestion de fichiers

Nous présentons la mise en œuvre en GAMMA d'un système de gestion de fichiers. Cette réalisation s'inspire de la spécification d'un système de fichiers UNIX présentée en [2]. Cette spécification exprime le comportement du système au niveau "requêtes" mais reste abstraite en ce qui concerne les problèmes de représentation des données ou de mise en œuvre. Il s'agit donc ici d'une mise en œuvre de "haut niveau" du système de gestion de fichiers.

## 4.1 Services offerts

Les services offerts par le système sont les suivants :

- création de fichier,
- ouverture de fichier,
- lecture et écriture de fichiers,
- fermeture de fichier.

Nous ne considérons pas dans cette mise en œuvre, les problèmes liés à la protection des fichiers, ni le cas des fichiers spéciaux.

Dans le paragraphe suivant, nous présentons l'architecture du système, et par la suite nous donnons un aperçu des programmes GAMMA mettant en œuvre les différents processus qui composent le système.

## 4.2 Architecture du système

Le système de gestion de fichiers contient quatre composants essentiels:

- le gestionnaire des requêtes relatives aux fichiers (FS), chargé de mettre en œuvre les différentes requêtes offertes par le système de gestion de fichiers,
- le système de nommage (NS) qui à un nom de fichier permet d'associer un identificateur de fichier,
- le système de stockage (SS) qui permet la consultation et la mise à jour des fichiers,

- le système de canaux (CS) chargé de gérer l'accès séquentiel aux fichiers.

Chaque composant est mis en œuvre par un processus GAMMA. Ces processus communiquent entre eux grâce au noyau $P_{sys}$.

## 4.3 Mise en œuvre

Plutôt que de détailler le rôle de chaque sous-système dans la mise en œuvre de chaque requête fichier, nous choisissons deux requêtes particulières, en l'occurrence l'ouverture de fichier et l'écriture dans un fichier, et montrons comment interfèrent les différents sous-systèmes.

### 4.3.1 Création de fichier (CREATE)

L'appel de la commande CREATE a pour but la création d'un nouveau fichier dans le système. Comme toutes les requêtes, elle est adressée au gestionnaire de requêtes aux fichiers (FS). Ce système constitue en effet l'interface entre le système de gestion de fichiers et les processus souhaitant accéder à ses services.

Sa mise en œuvre est décrite par un programme qui prend en argument :

- un multiensemble boîte aux lettres $BAL$
- un multiensemble contexte $CTX$ utilisé pour sauvegarder les paramètres d'une requête au moment de son exécution.

Le corps de ce programme est composé d'un ensemble de réactions que l'on peut partitionner suivant les requêtes que ces paires sont censées mettre en œuvre.
En ce qui concerne l'appel de la requête qui nous intéresse, ici CREATE, il consiste à placer dans la boîte aux lettres du gestionnaire de requêtes un triplet de la forme *(create, exp, name)*, *exp* étant l'identificateur du processus demandeur et *name* le nom du fichier dont la création est souhaitée. A la suite de quoi, le gestionnaire est en mesure d'effectuer le traitement de la requête en suivant la démarche décrite ci-après :

- Association d'un identificateur au nom du fichier,
- Création d'un canal et de la structure de données associée au fichier,
- Envoi du résultat au processus demandeur.

### 4.3.2 Association d'un identificateur au nom de fichier

La réaction qui permet au gestionnaire de requêtes de traiter la requête CREATE est la suivante :

```
rp (create,exp,name):BAL
by (send,NS,msg):BAL, (create,exp,name):CTX
    where
    msg = (createNS,FS,name)
```

Dans un premier temps, nous pouvons constater la prise en compte dans la boîte aux lettres du triplet *(create,exp,name)*. L'action associée consiste à envoyer un message de type *createNS* au système de nommage afin qu'il associe à *name* un identificateur de fichier.
Le système de nommage permet en fait au système d'accéder aux fichiers par leur nom. La structure du programme mettant en œuvre ce système est la suivante :

```
arg = ⟨BAL,MNS,MFID⟩
P_NS(arg) = (NS_OUT ∘ NS_IN)(⟨BAL,MNS,MFID⟩)
    where
    BAL = head(arg)
    MNS = head(tail(arg))
    MFID = head(tail(tail(arg))
```

Le multiensemble *MFID* contient les identificateurs de fichiers. Le multiensemble *MNS* contient une paire de la forme *(fid,name)* pour chaque fichier stocké, *fid* étant l'identificateur du fichier et *name* son nom. La requête *createNS* consiste à rechercher une paire *(fid, name)* dans MNS à partir d'un nom donné et à renvoyer l'identificateur correspondant au processus demandeur. Si cette paire n'existe pas, le système doit la créer. Le problème posé par l'existence de ces deux alternatives est résolu par le fait que le programme NS soit la composition de deux programmes GAMMA, NS_IN et NS_OUT.
Dans le corps de NS_IN, exécuté d'abord, apparaît la réaction suivante :

```
rp (createNS,exp,name):BAL, (name,fid):MNS
by (send,exp,msg):BAL, (name,fid):MNS
    where
    msg = (done_createNS,NS,(name,fid))
```

Elle exprime la prise en compte de la requête *createNS* à condition que le fichier existe déjà, c'est à dire que *MNS* contienne une paire *(name, fid)*.

L'action associée consiste alors à éliminer de la boîte aux lettres le message de requête *createNS* et d'y introduire une demande d'envoi de message à destination du demandeur, ce message signifiant que la requête *createNS* a bien été exécutée, et transportant la donnée *(name, fid)*.

Si le fichier n'existe pas déjà au moment de la requête *createNS*, celle-ci n'est pas traitée par le programme NS_IN. Le triplet *(createNS, exp, name)* subsiste dans la boîte aux lettres de NS après exécution de NS_IN. Il est donc pris en compte par le programme NS_OUT :

```
rp (createNS,exp,name):BAL, fid:MFID
by (send,exp,msg):BAL, (name,fid):MNS
    where
    msg = (done_createNS,NS,(name,fid))
```

L'action associée consiste simplement à associer un identificateur de fichier *fid* libre au nom de fichier *name*, et à placer la paire *(name, fid)* dans *MNS*.

### 4.3.3 Création de fichier

A la suite du traitement de la requête *createNS* par le système de nommage, le gestionnaire de requêtes récupère dans sa boîte aux lettres le triplet *(done_createNS, NS, (name,fid))* et va donc s'adresser au système de stockage et au système de canaux, en leur communiquant la valeur de *fid*, afin qu'ils entérinent la création de fichier. Ceci est mis en œuvre par la réaction suivante :

```
rp (done_createNS,NS,(name,fid)):BAL, (create,exp,name):CTX
by (send,SS,msg1), (send,CS,msg2):BAL, (create,exp,name,fid):CTX
    where
    msg1 = (createSS,FS,fid)
    msg2 = (createCS,FS,fid)
```

Voyons à présent comment réagit le système de stockage.

#### Création de la structure de données associée au fichier

De manière similaire au système de nommage , le système de stockage est mis en œuvre par une composition de deux programmes GAMMA :

```
arg = ⟨BAL, MSS⟩
P_SS(arg) = (SS_OUT ∘ SS_IN) (BAL,MSS)
    where
    BAL = head(arg)
    MSS = head(tail(arg))
```

Le multiensemble *MSS* est utilisé comme moyen de stockage de fichiers. Il contient des paires de la forme *(fid, file)* représentant chacune un fichier dans le système, *fid* étant l'identificateur du fichier et *file* une liste d'octets contenant les informations du fichier. Le programme SS_IN comporte la réaction suivante :

```
rp (createSS,exp,fid):BAL (fid,file):MSS
by (fid,NIL):MSS, (sen, exp, msg):BAL
    where
    msg = (done_createSS, ss, fid)
```

Elle permet de traiter le cas où le fichier que l'on cherche à créer existe déjà. L'identificateur de fichier se voit alors associer une liste d'octets vide. Cela revient à "effacer" le fichier. Si le fichier n'existe pas, c'est le programme SS_OUT qui traite la requête *createSS* :

```
rp (createSS,exp,fid):BAL
by (fid,NIL):MSS , (send, exp, msg):BAL
    where
    msg = (done_createSS, SS, fid)
```

L'action consiste à placer dans *MSS* un nouveau couple comportant d'une part l'identificateur de fichier et d'autre part une liste d'octets vide. Dans les deux cas, un accusé de fin d'exécution est renvoyé au gestionnaire de requêtes. Il reste maintenant à voir comment la requête *createCS* est traitée par le système de canaux.

### Création d'un canal

Le programme décrivant le système de canaux possède la structure suivante :

```
arg = ⟨BAL, MCS, MCID⟩
P_CS(arg) = (CS_OUT o CS_IN) (BAL,MCS, MCID)
    where
    BAL = head(arg)
    MCS = head(tail(arg))
    MCID = head(tail(tail(arg)))
```

Le multiensemble *MCID* contient les identificateurs de canaux (descripteurs de fichiers).
Le multiensemble *MCS* contient des triplets de la forme *(cid, fid, pstn)* représentant les canaux :

- *cid* est l'identificateur du canal
- *fid* est l'identificateur du fichier associé au canal
- *pstn* est la position courante d'accès au fichier

A chaque création ou ouverture de fichier correspond la création d'un canal permettant l'accès au fichier. Le traitement de la requête va consister à déposer dans MCS un nouveau triplet *(cid, fid, pstn)* et rendre l'identificateur du canal alloué (*pstn* étant initialisé à zéro).

```
rp (createCS,exp,fid):BAL, cid:MCID
by (send,exp,msg):BAL, (cid,pstn,fid):MCS
    where
    msg = (done_createCS,CS,(fid,cid))
    pstn = 0
```

Pour terminer la création d'un fichier, le gestionnaire de requêtes aux fichiers doit prendre en compte les acquittements provenant des différents systèmes avec lesquels il a communiqué.

```
rp (done_createCS,CS,(fid,cid)):BAL, (done_createSS, SS, fid):BAL,
   (create,exp,name,fid):CTX
by (send,exp,msg):BAL
    where
    msg = (done_create,FS,(name,cid))
```

Outre un message indiquant la fin du traitement de la requête *create*, le gestionnaire communique au processus demandeur l'identificateur du canal créé.

Le lecteur intéressé pourra trouver en annexe de ce document les programmes complets mettant en œuvre les différents composants intervenant dans la gestion des fichiers.

Nous nous contentons ici de décrire brièvement le traitement de l'écriture dans un fichier.

### 4.3.4 Ecriture de fichier

La requête WRITE permet l'écriture de données dans un fichier. A l'appel de cette requête, le processus demandeur envoie l'identificateur *cid* du canal associé au fichier qu'il souhaite mettre à jour et une liste *data* contenant les données à écrire.
Cette requête est mise en œuvre par l'enchaînement des tâches suivantes :

- Demande de l'identificateur *fid* de fichier et de la position courante d'accès associés à l'identificateur de canal fourni,
- Demande de lecture du fichier associé à l'identificateur de fichier reçu,
- Après réception du fichier :
  - recopie des données fournies en argument de la requête, dans la liste constituant le fichier, à partir de la position indiquée,
  - demande d'écriture du fichier modifié,
  - mise à jour de la position courante d'accès au fichier par le canal considéré.

Les réactions décrivant cette requête sont les suivantes :

```
rp (write,exp,(cid,data)):BAL
by (send,CS,msg):BAL, (write,exp,cid,data):CTX
    where
    msg = (look_upCS,FS,cid)

rp (done_lookupCS,CS,(cid,pstn,fid,rep)):BAL,
   (write,exp,cid,data):CTX
by  msg:BAL, save:CTX
```

```
    where
    msg = if rep = Ok then {(send,SS,msg1)}
          else {(send,exp,msg2)}
    save = if rep = Ok then {(write,exp,cid,data,fid,pstn)}
           else ∅
    msg1 = (look_upSS,FS,fid)
    msg2 = (done_write,FS,(cid,NoSuchCid))

rp (done_look_upSS,(fid,file)):BAL, (write,exp,cid,data,fid,pstn):CTX
by (send,exp,msg1),(send,SS,msg2),(send,CS,msg3):BAL
    where
    msg1 = (done_write,FS,(cid,Ok))
    msg2 = (putSS,FS,(fid,file'))
    msg3 = (putCS,FS,(cid,pstn',fid))
    file' = modif(file,data',pstn)
    data' = zero append data
    zero = cons_l0(pstn - long(file))
    pstn' = pst + long(zero) + long(data)
    cons_l0 (c) = if c ≤ 0 then NIL else cons(0,cons_l0(c-1))
```

La fonction *modif(x,y,p)* qui apparaît dans la troisième réaction est utilisée pour remplacer les éléments de la liste $x$ à partir de la position $p$, par les éléments de la liste $y$. Le nombre d'éléments remplacés est égal à la longueur de la liste $y$. La liste *data'* est formée de l'union des listes *zero* et *data*. La liste *data* contient les données à écrire. La liste *zero* contient des octets ayant la valeur de 0. Si la position d'écriture possède une valeur inférieure ou égale à la taille du fichier, la liste *zero* est vide, sinon, celle-ci possède une longueur égale à $pstn - long(file)$. $long(file)$ représente la taille du fichier. Le message *msg1* sert à avertir le processus demandeur de la terminaison de la requête. Le message *msg2* contient une demande de sauvegarde du fichier modifié (*putSS*). Le message *msg3* représente une demande de sauvegarde du canal. La nouvelle position d'accès au fichier est *pstn'*.

# 5 Programmation de quelques commandes UNIX

Nous décrivons ici la programmation GAMMA d'un interpréteur de commandes (*shell*) simplifié, d'une commande simple de listage de fichiers (*ls*). Nous indiquons ensuite comment peut être traité le problème des commandes génériques. Et enfin, nous présentons la programmation d'un utilitaire important de maintenance de fichiers (*make*). A partir de ces différents exemples, nous montrons les avantages du formalisme GAMMA pour l'expression du parallélisme permis par l'exécution de ces commandes.

## 5.1 Un *shell* simplifié

Le code simplifié d'un interpréteur de commandes dans un langage impératif est:

```
while TRUE {
    reading(command,parameters,bkg);
    n = fork ();
    if n ≠ 0
       then if not bkg then wait(&status)
       else exec(command,parameters)
   }
```

Après lecture d'une commande et de ses paramètres, la requête *fork* est utilisée pour créer un processus fils chargé d'exécuter cette commande. Lors de l'exécution de la requête *fork* par le système, une nouvelle copie du *shell* est créée. La valeur rendue après l'exécution de la requête *fork* est utilisée pour distinguer le code des deux processus. Le processus père reçoit l'identificateur du processus créé et le processus fils la valeur zéro. La commande est exécutée par le processus fils par le biais de la requête *exec*. Cette primitive remplace le code et les données du processus appelant par ceux correspondant à la commande. Si l'exécution n'est pas en arrière plan, le processus père attend simplement la terminaison du fils au moyen de la requête *wait*. Le code mettant en œuvre la commande doit être terminé par la requête *exit(etat)*. L'exécution de cette requête entraîne la disparition du processus fils et le réveil du processus père. La valeur de la variable *etat* est reçue par le processus père dans la variable *status*. Après l'exécution de la commande, le *shell* est prêt à lire d'autres commandes. Si l'exécution est en arrière plan, le processus père n'attend pas la terminaison de la commande et il est prêt à recevoir d'autres commandes.

Le programme GAMMA mettant en œuvre le *shell* est le suivant :

```
P_shell (⟨BAL_SH,CTX_SH⟩) = ⟨BAL_SH',CTX_SH'⟩
    where
    (BAL_SH',CTX_SH') = sh(BAL_SH,CTX_SH)
    sh(BAL,CTX) = (gam BAL,CTX -> BAL,CTX

    rp (command,exp,parameters,bkg):BAL
    by (fork,pid_sh):BAL, (command,exp,parameters,bkg):CTX

    rp (done_fork,pid1),(pid_fils,pid2):BAL
       (command,exp,parameters,bkg), (pid_processus,pid_sh):CTX
    if (pid1 = 0)
    by (exec,command,arg'',pid2):BAL
        where
        arg'' = ⟨BAL_FILS,CTX_FILS⟩
        BAL_FILS = bag((command,parameters))
        CTX_FILS = bag((pid_père,pid_sh)) ∪ bag((pid_processus,pid2))

    rp (done_fork,pid1):BAL, (command,exp,parameters,bkg):CTX
    if (pid1 ≠ 0 and bkg)
    by (send,exp,msg):BAL
        where
        msg = (done_com_bkg,pid_sh)

    rp (done_fork,pid1), (done_com,pid1,etat):BAL
```

```
   (command,exp,parameters,bkg):CTX if (pid1 ≠ 0 and not bkg)
by (send,exp,msg):BAL
    where
    msg = (done_com,pid_sh,(command,etat))

)((BAL,CTX))
```

Le programme est divisé en quatre parties:

- lecture de la commande et demande de création du processus fils,
- exécution de la commande par le fils au moyen de la requête *exec*,
- envoi du message de *commande en cours d'exécution* dans le cas d'une exécution en arrière plan,
- attente du père de la fin d'exécution de la commande et envoi du résultat de l'exécution au processus générateur de la commande.

Les valeurs rendues par la requête *fork* (*(done_fork,pid1)*) ont la même signification que sous UNIX. Le processus fils reçoit de plus la paire *(pid_fils,pid2)* indiquant son identificateur de processus. Comme *CTX_SH* après l'exécution du *fork* est le même pour le processus père et le processus fils, ce dernier peut connaître l'identificateur de son père *pid*. L'exécution de la commande est faite par la requête *exec*. L'exécution de cette requête par le système entraîne le remplacement du triplet représentant le fils par le triplet *(P_command,arg",pid2)*, où *P_command* est le code de la commande stocké dans un fichier nommé *command*, et où *arg"* est une liste formée des multiensembles *boîte aux lettres BAL_FILS* et *contexte CTX_FILS*. *BAL_FILS* contient le n_uplet demandant le service et *CTX_FILS* contient les identificateurs des processus fils et père.

Dans le cas d'une exécution en arrière plan (*bkg* = *TRUE*), il ne faut pas attendre la terminaison de la commande. C'est pourquoi le message *done_com_bkg* est envoyé au processus lecteur des commandes, afin qu'il puisse envoyer la commande suivante. Dans le cas où il faut attendre la terminaison, le processus père reste bloqué en attendant le message de terminaison envoyé par le fils. Il faut noter que nous n'utilisons pas la requête *wait* du fait que nous ne pouvons pas garantir son traitement avant celui de la requête *exit* émise par le processus fils lors de sa terminaison.

## 5.2 Traitement de la commande ls

La commande *ls* est utilisée pour faire afficher la liste des noms de fichiers appartenant à un répertoire. Le programme mettant en œuvre cette commande cherche dans un multiensemble les noms appartenant à un répertoire donné. Dans ce multiensemble les noms sont stockés sous la forme d'une paire ayant la forme *(name,fid)* où *fid* est l'identificateur du fichier et *name* son nom.

Le nom d'un fichier est formé d'une séquence de syllabes séparées par le caractère /:

$name = /syl_1/.../syl_n$

La séquence $/syl_1/.../syl_{n-1}$ représente le répertoire auquel appartient le fichier. L'exécution de la requête *ls* entraîne donc la recherche des fichiers dont les noms commencent par la séquence $/syl_1/.../syl_{n-1}$ représentant le répertoire demandé. Une liste est ainsi construite, contenant toutes les dernières syllabes ($syl_n$) de ces noms de fichiers.

Nous disposons de la fonction *last* qui lors de l'application à un nom *name* rend sa dernière syllabe, et de la fonction *front* qui appliquée à un nom *name* rend la séquence de syllabes $/syl_1/.../syl_{n-1}$ du nom.

La commande est codée sous la forme *(ls,l_dir)*, où *l_dir* est une liste contenant les répertoires dont nous souhaitons faire afficher le contenu.

Cette commande est mise en œuvre par le programme suivante:

```
P_ls(⟨BAL_LS,CTX_LS⟩) = ⟨BAL_LS',CTX_LS'⟩
(BAL_LS',CTX_LS') = ( gam BAL,CTX -> BAL,CTX

   rp (ls,l_dir):BAL, (pid_processus,pid):CTX
   by (send,sys,msg):BAL, (ls,l_dir), (pid_processus,pid):CTX
       where
       msg = (get_arg,pid,pid_ns)

   rp (done_get_arg,sys,arg):BAL, (ls,l_dir), (pid_processus,pid),(pid_père,pid'):CTX
   by (send,pid',msg1), (send,tty,msg2), (exit,pid,ok):BAL
       where
       msg1 = (done_com,pid,ok)
       msg2 = (print,files)
       files = unbag(FILES)
       FILES = search(NS,l_dir)
       NS = head(tail²(arg))
       search(N,l_d) = (gam L -> X -> Y
          rp l:L
          if tail(l) ≠ nil
          by tail(l):L, head(l):X

          rp dir:X
          by ⟨(dir,tri(⟨l_fich⟩))⟩:Y
             where
             ⟨l_fich⟩ = unbag(chercher(dir,N))
             chercher(d,N) = (gam N -> N
                rp (name,fid):N
                if (front(name) = d)
                by ⟨last(name)⟩:N

                rp l_fich,l_fich':N
                by l_fich append l_fich':N
             )N

          rp l_dir_fich, l_dir_fich' : Y
```

```
by l_dir_fich append l_dir_fich' : Y
)(bag(l_d))
```

Ce programme est composé de deux réactions. La première est utilisée pour demander au système une copie de l'argument du processus mettant en œuvre le système de nommage (*pid_ns*) du système gestionnaire de fichiers. Cela est dû au fait que le processus mettant en œuvre la commande *ls* nécessite le multiensemble *NS* contenant les noms de tous les fichiers dans le système.

Après avoir obtenu l'argument demandé et avoir cherché les noms de fichiers des répertoires sollicités, la deuxième réaction est utilisée pour:

- communiquer au *shell* la terminaison de la commande (*msg1*),
- demander au gestionnaire du terminal l'impression des noms de fichiers (*msg2*),
- émettre la requête *exit* demandant la disparition du processus.

Il faut remarquer l'obtention de la liste *files* contenant les noms de fichiers de chaque répertoire sollicité. Cette liste est construite à l'aide de l'expression *search(NS,l_dir)*. Cette expression est un programme GAMMA appliqué à un multiensemble contenant la liste des répertoires. Le programme cherche en parallèle tous les noms dans *NS* appartenant à chaque répertoire de la liste *l_dir*. Le multiensemble résultat a la forme:

`{(dir_1,l_f_1), ..., (dir_n,l_f_n)}`

Chaque `dir_i` est un répertoire de la liste *l_dir*, et chaque `l_f_i` est une liste contenant les noms des fichiers dans le répertoire `dir_i`. Chaque liste `l_f_i` est construite à l'aide de l'expression *chercher(dir,NS)*. Cette expression est un programme GAMMA appliqué au multiensemble *NS*. Le programme cherche dans *NS* les noms qui appartiennent au répertoire *dir*.

Grâce à l'imbrication des deux programmes GAMMA dans la deuxième réaction du processus, nous avons pû exprimer la recherche en parallèle des noms de fichiers appartenant à tous les répertoires sollicités.

## 5.3 Traitement de commandes génériques

Un exemple de commande générique est *ls *.c*. Celle-ci demande la liste de tous les noms de fichiers se terminant par la chaîne *.c*. Cette commande est codée sous la forme de la paire *(ls_f*,(ter,dir))* où *ter* est la chaîne terminant les noms cherchés et *dir* le répertoire où il faut les rechercher.

La mise en œuvre de cette commande est similaire à celle de la commande que nous avons analysée dans le paragraphe précédent, mis à part le code du programme servant à

construire la liste triée *files* qui contient les noms de fichiers terminés par la chaîne *ter* dans le répertoire *dir*:

```
files = tri(unbag(FILES))
FILES = search (NS,dir,ter)
search(N,d,t) = (gam N -> N'
   rp (name,fid):N
   if (front(name) = d) and (end(last(name),t)
   by (last(name)):N'

   rp (name), (name'): N'
   by (name) append (name') : N'
) N
```

Le programme *search* cherche dans *NS* tous les noms *name* appartenant au répertoire demandé (*front(name) = dir*) et terminés par la chaîne *ter* (*end(last(name),ter)*). La fonction *end* teste si la syllabe *last(name)* est terminée par la chaîne *ter*.

Nous pouvons constater dans ces deux derniers exemples la simplicité de la programmation et la puissance dans l'expression du parallélisme offert par le formalisme GAMMA.

## 5.4 Un utilitaire : make

Pour chaque fichier, le système conserve l'information de "date de la dernière modification". Le programme utilitaire *make* lit une spécification des dépendances de programmes entre eux et l'interprète pour créer une nouvelle version compilée du fichier. Il vérifie les dates de dernière modification pour réaliser le minimun nécessaire de compilations. *make* est particulièrement intéressant lorsque le programme est suffisamment important pour être composé de plusieurs fichiers. Les spécifications des dépendances d'un programme doivent se trouver dans un fichier de nom *makefile*.

Exemple de contenu de fichier *makefile*:

```
prog : prog.o init.o
       cc prog.o init.o -o prog
```

Cela signifie que *prog* depend de *prog.o* et *init.o*, et que *prog.o* est converti en *prog* à l'aide du compilateur C. *prog* est dit "entrée de *makefile*". Un fichier *makefile* peut contenir plusieurs entrées.

Nous souhaitons décrire à l'aide de GAMMA le traitement de la commande: *make nom_prog* où *nom_prog* doit être le nom d'une entrée du fichier *makefile*.

La commande *make nom_prog* est prise en compte par le shell, qui fait en sorte de placer dans le système un nouveau processus décrit par *(Pmake, arg, id_mk)* avec:

- *Pmake* le programme mettant en œuvre la commande *make*,
- *arg* une liste contenant les multiensembles:
  - *BAL* boîte aux lettres associée au processus *Pmake* qui contient au départ *nom_prog*,
  - *CTX,M,M',N,R* multiensembles de travail.
- *id_mk* l'identificateur du processus.

La première phase du programme va consister à construire à partir du fichier *makefile* présent dans le répertoire courant, un multiensemble $N$ de triplets *(c,D,P)* donnant pour chaque fichier $c$ impliqué dans la mise à jour du fichier *nom_prog*, l'ensemble $D$ des fichiers dont il dépend et le procédé $P$ de mise à jour. Les fichiers sources ont un champ $D$ égal à l'ensemble vide.

Exemple: si le fichier *makefile* contient :

```
n1  :  n2 n3
       p1
n2  :  n4 n5
       p2
m1  :  m2 m3      (* entrée m1, dependance {m2,m3}, procédé q1 *)
       q1
m2  :  m4 m5 m6
       q2
m6  :  m7
       q3
```

alors la commande *make m1* entraîne la construction du multiensemble $N$ suivant :

```
{
(m1, {m2,m3},q1),
(m2, {m4,m5,m6}, q2),
(m6, {m7}, q3),
(m3, {}, skip),
(m4, {}, skip),
(m5, {}, skip),
(m7, {}, skip),
}
```

Pour aboutir à ce multiensemble $N$, la démarche est celle-ci. L'entrée *m1* est recherchée dans le fichier *makefile*. L'ensemble de ses dépendances est {*m2,m3*}, le procédé de mise à jour *q2*. Donc le triplet *(m1,{m2,m3},q1)* est introduit dans $N$. Ensuite on fait de même avec

les fichiers *m2* et *m3* jusqu'à aboutir à des triplets dont le deuxième champ est l'ensemble vide.

La deuxième phase consiste en la mise à jour proprement dite des fichiers.

Voici le texte du programme GAMMA réalisant le programme *Pmake*.

```
Pmake = (gam(BAL,CTX,M,M',N,R) -> (BAL,CTX,M,M',N,R)

   rp (make,nom_prog):BAL
   by (send,FS,msg1),(send,FS,msg):BAL, (make,nom_prog):CTX
       where
       msg1 = (open,id_mk,Makefile)
       msg2 = (fstat,id_mk,Makefile)

   rp (done_open,FS,(Makefile,cid)), (done_fsta,FS,(Makefile,etat):BAL
   by (send,FS,msg):BAL
       where
       msg = (read,id_mk,(cid,etat.length))

   rp (done_read,FS,(cid,liste_octets)):BAL, (make,nom_prog):CTX
   by (elimdoubles o triplets_out o triplets_in o analyse)
      (liste_octets,nom_prog):N
       where

       (*(1)
       analyse est une fonction, qui étant donnée une liste d'octets représentant un
       fichier makefile, rend la paire (FMAKEFILE,nom_prog) où FMAKEFILE est un
       multiensemble de triplets de la forme (c,D,P), un triplet particulier
       correspondant à une règle de makefile (c est le nom d'un fichier, D est
       l'ensemble des fichiers dont dépend c, P est le procédé de mise à jour).*)

       (*2*)
       (triplets_in) (FMAKEFILE,nom_prog) =
       (gam FMAKEFILE -> FMAKEFILE
          rp (prog',D',P'):FMAKEFILE, (make, nom_prog):CTX
          if prog' = nom_prog
          by (prog,D',P'):N, D':Md

          rp d:Md, (d',D',P'):FMAKEFILE
          if d = d'
          by (d,D',P'):N, D':Md
       ) FMAKEFILE

       (* Ce programme rend un multiensemble FMAKEFILE' contenant tous les triplets
       de FMAKEFILE associés aux fichiers intervenant dans la mise à jour du fichier
       nom_prog. *)

       (*3*)
       (triplets_out) (FMAKEFILE') =
       ( gam Md -> N
          rp d:Md
          by (d,∅,skip):N
       )FMAKEFILE'
```

```
    (* Ce programme rajoute dans N, pour chaque fichier "source" et impliqué, un
    triplet (d,{},skip).*)

    (*4*)
    (elimdoubles) N =
    (gam N -> N
       rp (c,D,P):N, (c',D',P'):N
       if c = c' and D = D' and P = P'
       by (c,D,P):N
    )N

    (* Ce programme élimine de N les occurrences multiples de triplets. *)

(* Le multiensemble N une fois construit, les réactions suivantes mettent en
œuvre le reste du programme Pmake. *)

rp (c,D,P):N
by (send,FS,msg):BAL, (c,D,P,D,false):M'
    where
    msg = (fstat,id_mk,c)

rp (done_fstat,FS,(name,etat)):BAL, (c,D,P,d,b):M'
if (name = c)
by (c,D,P,d,b,etat.date):M

rp (c,D,P,d,b,date):M
if (d = ∅) and (not b)
by (c,date):R

rp (c,D,P,d,b,date):M, (r,date'):R
if (r in d) and (date' ≤ date)
by (c,D,P,d-c,b,date):M, (r,date'):R

rp (c,D,P,d,b,date):M, (r,date'):R
if (r in d) and (date' > date)
by (c,D,P,d-c,true,date):M, (r,date'):R

rp (c,D,P,d,b,date):M
if (d = ∅) and (not b)
by (c,D,P,b,date):CTX, (fork,id_mk):BAL

rp (done_fork,pid),(done_P,x,c):BAL
if (pid ≠ 0)
by (send,FS,msg):BAL
    where
    msg = (fstat,id_mk,c)

rp (done_fstat,FS,(name,etat)):BAL, (c,D,P,b,date):CTX
if (name = c)
by (c,etat.date):R

rp (done_fork,pid),(pid_fils,pid'):BAL, (c,D,P,b,date):CTX
if (pid = 0)
by (exec,P,arg,pid'):BAL
    where
```

```
arg = ⟨BAL'⟩
BAL' = D ∪ (pid_père,id_mk),(pid_fils,pid')
```

A partir des triplets *(c,D,P)* de *N*, on construit des n_uplets de la forme *(c,D,P,D,false,date)*, la date étant fournie par le système de gestion de fichier et correspondant à la date de la dernière modification du fichier *c*.

Le principe du programme consiste à construire progressivement le multiensemble *R* des fichiers mis à jour. Considérons un n_uplet *(c,D,P,d,b,date)*, *D* représentant l'ensemble des dépendances du fichier *c*, et *d* étant initialisé à *D*. Lorsqu'on constate la présence dans *R* d'un fichier qui apparaît dans *d*, on l'enlève de *d*. Il est donc clair que lorsque $d = \{\}$, toutes les dépendances de *c* ont été traitées et *c* peut à son tour être mis à jour.

On peut noter que les fichiers sources peuvent immédiatement être placés dans *R*.

Le booléen *b* indique s'il faut ou non appliquer à *c* le procédé de mise à jour. Il est mis à vrai lorsqu' en "vidant" *d*, on trouve un fichier plus "récent" que *c*.

Le procédé *P* pouvant être une commande quelconque, son application au fichier *c*, passe par la création d'un processus chargé de son exécution (*fork* puis *exec*).

De même que pour les commandes traitées précédement, nous constatons que cette programmation de *Pmake* permet d'engendrer du parallélisme à l'exécution , aussi bien lors de la constitution du multiensemble de triplets que lors de la mise à jour effectuée des fichiers.

Un problème reste à préciser en ce qui concerne la terminaison de ce programme. Toute commande doit en effet renvoyer une signal de fin d'exécution au processus *shell*. Nous avons omis ici de détailler ce point afin de ne pas alourdir la programmation.

# 6 Discussion

Nous avons présenté dans ce document, la programmation en GAML des différents constituants d'un noyau UNIX: gestion des processus, gestion des fichiers, interpréteur de commandes. L'étude réalisée nous permet de dégager un certain nombre de points illustrant les intérêts et particularités de la programmation système en GAMMA.

- Expression simple du parallélisme.
  Le modèle GAMMA permet l'exécution en parallèle de différentes réactions composant un programme, à condition qu'elles fassent intervenir des ensembles disjoints d'éléments du multiensemble. Cette propriété nous permet d'exprimer simplement l'exécution parallèle de différents processus GAMMA au moyen du noyau $P_{sys}$. Nous utilisons également cette propriété du modèle GAMMA pour mettre en œuvre des processus serveurs traitant en même temps les requêtes de différents processus clients.

- Gestion de ressources critiques.
  Le problème de gestion de ressources critiques a donné lieu à différentes propositions d'outils de synchronisation entre processus tels que les sémaphores,les moniteurs etc.. Ces outils permettent un accès atomique aux ressources critiques. Ils empêchent l'accès à une ressource si celle-ci est déjà occupée. Lorsque nous programmons en utilisant le formalisme GAMMA nous n'avons pas besoin de ces outils car c'est la saisie atomique des n_uplets dans un multiensemble qui assure la synchronisation entre les processus. Dans [1], les auteurs traitent par exemple le problème des philosophes. Ils montrent une façon simple d'exprimer à l'aide de GAMMA, le problème d'allocation des fourchettes. Nous reprenons ici cet exemple.

  Le problème consiste à montrer un algorithme qui assure le non-interblocage entre les les philosophes et garantit que n'importe quel philosophe souhaitant manger acquiert ses deux fourchettes au bout d'un temps fini.

  L'état du système est représenté par le multiensemble $FK$ contenant des objets correspondant aux fourchettes disponibles et aux philosophes en train de manger. Le multiensemble initial $FK = \{F_0, F_1, F_2, F_3, F_4\}$ contient cinq fourchettes; si le philosophe $P_i$ est autorisé à manger, le multiensemble $FK$ devient alors $(FK - \{F_i, F_{i\oplus 1}\}) \cup \{P_i\}$ où $\oplus$ représente l'addition modulo cinq. Quand $P_i$ a fini de manger, il relâche ses deux fourchettes.

  Le programme GAMMA qui exprime cet algorithme est

```
FK = {F0, F1, F2, F3, F4}
philosophes (FK) = (gam FK -> FK
   rp Fi, Fi⊕1:FK
   by Pi:FK

   rp Pi:FK
   by Fi, Fi⊕1:FK
   )(FK)
```

  La saisie atomique des fourchettes $F_i$ et $F_{i\oplus 1}$ empêche l'interblocage éventuel des philosophes $P_i$ et $P_{i\oplus 1}$. L'existence d'un choix équitable garantit l'absence de famine.

- Synchronisation implicite de l'exécution des processus.
  La synchronisation d'un processus avec un nombre quelconque d'autres processus est facile à exprimer à l'aide de GAMMA. Il suffit de l'indiquer dans la condition de réaction. Par exemple, un processus qui demande trois ressources à trois processus peut le faire par:

```
rp ...
by (send,P1,(get, ressource1)),(send,P2,(get, ressource2)),
   (send,P3,(get, ressource3)):BAL
```

  La condition de réaction qui exprime l'attente de ces ressources est:

```
rp (done_get,ressource1),(done_get,ressource2),
   (done_get,ressource3):BAL
if ...
```

- Communication asynchrone entre processus.
  Du point de vue théorique, un multiensemble n'est pas limité en taille. Cela nous permet d'utiliser les multiensembles comme "moyen de communication". C'est le rôle des multiensembles appelés boîte aux lettres dans notre système de processus communicants. Ainsi la boîte aux lettres peut être considérée comme un tampon de longueur infinie qui sert à mettre en œuvre une communication asynchrone entre les processus du système. Cependant les messages stockés dans la boîte aux lettres ne sont pas traités dans un ordre déterminé, cela est dû au non déterminisme du modèle GAMMA.

- Exécution séquentielle de certaines actions des processus.
  Lorsque l'activité d'un processus se décompose en plusieurs étapes qui doivent être exécutées les unes à la suite des autres, il suffit de concevoir des programmes GAMMA réalisant chacune de ces étapes et d'effectuer ensuite la composition de ces programmes. La composition de programmes GAMMA permet donc d'exprimer simplement la sérialisation de certaines actions des processus.

- Homogénéité de la programmation obtenue
  A la lecture de ce document, nous pouvons constater que le seul outil utilisé pour la programmation du système est le langage GAML. Quel que soient les aspects abordés, exécution, synchronisation, communication, création ou mort de processus, tous sont exprimés en utilisant le même formalisme. Il n'est jamais fait mention d'élément extérieur. Il en résulte donc une grande homogénéité et une forte cohérence dans la programmation obtenue.

De toutes ces constatations, nous pouvons conclure que le formalisme GAMMA offre beaucoup d'attrait pour la programmation système, en particulier lorsqu'il s'agit d'exprimer le parallélisme d'exécution, la synchronisation, la communication et la coopération entre processus.

En outre la programmation de systèmes en GAMMA nous semble plus naturelle que la programmation faite en langage fonctionnel. Cela est dû au parallélisme et au non déterminisme implicites du modèle.

En effet, les applications système nécessitant l'expression du non déterminisme, une primitive spéciale est indispensable. Une approche [3] a consisté en l'utilisation de la primitive *merge* qui occasionne la perte d'une importante propriété des langages fonctionnels: *la transparence référentielle.* Afin d'éviter l'utilisation de ce type de primitive, une autre approche [6] et [8] a visé la modification de la machine abstraite qui met en œuvre le langage fonctionnel. Les processus continuent à être des fonctions mais la prise en compte du non déterminisme reste cachée dans la machine abstraite. En outre, cette machine est utilisée pour exécuter en parallèle tous les processus du système et pour gérer la communication entre les processus.

Ainsi le noyau de ces systèmes n'est pas programmé dans le même langage fonctionnel que le reste du système.

Dans l'état actuel de nos travaux, nous avons montré qu'il était possible à l'aide de GAMMA de décrire un noyau UNIX. D'une manière générale, il est clair que ce formalisme peut être utilisé comme outil de spécification de systèmes. Reste à savoir si à partir d'une telle spécification d'un système, il est possible d'obtenir sans trop de difficultés une mise en œuvre réaliste de ce système.
C'est le problème que nous allons maintenant tenter de résoudre. Nous allons étudier les possibilités de faire subir à des programmes GAMMA des transformations, de manière à se rapprocher le plus possible d'une mise en œuvre de ces programmes.

# Bibliographie


[1] Banâtre J.-P., Le Métayer D.. Programming by Multiset Transformation. *A paraître dans les Communications de l'ACM.*

[2] Carroll Morgan, Bernarfd Sufrin. Specification of the UNIX Filing System.*Specification Case Studies. Prentice Hall*

[3] S.B.Jones, A.Sinclair. Functional Programming and Operating Systems. *The Computer Journal, vol. 32, No.2 1989.*

[4] B. Kernigan, R. Pike. L'environnement de Programmation UNIX. *InterEditions*

[5] H.Lucas, B.Martin, G. de Sablet. UNIX Mécanismes de base, Langage de commande, Utilisation. *Eyrolles*

[6] William Stoye. A new scheme for writing functional Operating Systems.*Computer Laboratory. Technical Report No. 56, 1984 University of Cambridge.*

[7] Andrew S. Tanenbaum. Operating Systems: Design and Implementation. *Prentice Hall*

[8] D.A. Turner. Functional Programming and Communicating Processes. *Proc. PARLE 87. LNCS 259, Springer Verlag (1987).*

# A Programme du Gestionnaire de Processus

```
P_SYS = (gam SYS,BAL,BLK,PIDS,PROC -> SYS,BAL,BLK,PIDS,PROC
```

## A.1 Exécution de processus

```
rp (P,arg,pid):SYS
by (P,arg'',pid):SYS, req:BAL
    where
    bal = head(arg)
    arg' = P(arg)
    bal' = head(arg')
    req = bal' - bal
    bal''= bal ∩ bal'
    arg''= cons(bal'',tail(arg'))
```

## A.2 Communication entre processus

```
rp (send,pid,msg):BAL, (P,arg,pid):SYS
by (P,arg',pid):SYS
    where
    arg' = cons(bal',tail(arg))
    bal' = head(arg) ∪ {msg}

rp (send,sys,msg):BAL
by  msg:BAL
```

## A.3 Création de processus

```
rp (fork,pid):BAL, (P,arg,pid):SYS, pid':PID,
   (pid,pid_père,l_fils,wait):PROC
by (P,arg',pid),(P,arg'',pid'):SYS,(send,FS,msg):BAL
   (pid,pid_père,l_fils',wait):PROC,(pid',pid,l_fils'',wait'):PROC
    where
    arg' = cons(bal',tail(arg))
    bal' = head(arg) ∪ {msg1}
    msg1 = (done_fork,pid')
    arg'' = cons(bal'',tail(arg))
    bal'' = {msg2} ∪ {msg3}
    msg2 = (done_fork,0)
    msg3 = (pid_fils,pid')
    msg = (do_fork,sys,(pid,pid'))
    l_fils' = cons(pid',l_fils)
    l_fils''= NIL
    wait' = FALSE
```

## A.4 Attente de processus

```
rp (wait,pid):BAL, (P,arg,pid):SYS, (pid,pid_père,l_fils,wait):PROC
by (P,arg,pid):BLK, (pid,pid_père,l_fils,wait'):PROC
    where
    wait' = TRUE
```

## A.5 Terminaison de processus

```
rp (exit,pid,stat):BAL, (P,arg,pid):SYS, (pid,pid_père,l_fils,wait),
   (pid_père,pid_ancêtre,l_fils',wait'):PROC
if (wait' = FALSE) and (l_fils = NIL)
by (pid_père,pid_ancêtre,l_fils'',wait'):PROC, (send,FS,msg):BAL
    where
    msg = (do_exit,sys,pid)
    l_fils'' = delete(l_fils',pid)

rp (exit,pid,stat):BAL, (P,arg,pid):SYS, (P',arg',pid_père):BLK
   (pid,pid_père,l_fils,wait), (pid_père,pid_ancêtre,l_fils',wait'):PROC
if (wait' = TRUE) and (l_fils = NIL)
by (P',arg'',pid_père):SYS, (pid_père,pid_ancêtre,l_fils'',wait''):PROC
   (send,FS,msg):BAL
    where
    msg = (do_exit,sys,pid)
    arg'' = cons(bal'',tail(arg'))
    bal'' = head(arg') ∪ {msg}
    msg = (done_exit,pid,stat)
    l_fils''= delete(l_fils',pid)
    wait'' = FALSE

rp (exit,pid,stat):BAL,(pid,pid_père,l_fils,wait),
   (pid_fils,pid,l_fils',wait'):PROC
if (head(l_fils) = pid_fils)
by (exit,pid,stat):BAL,(pid,pid_père,l_fils'',wait),
   (pid_fils,INIT,l_fils',wait'):PROC
    where
    l_fils'' = tail(l_fils)
```

## A.6 Remplacement de code

```
rp (exec,nom_fich,argv,pid):BAL, (P,arg,pid):SYS
by (send,FS,msg1),(send,FS,msg2):BAL, (P,arg,pid,nom_fich,argv):BLK
    where
    msg1 = (open,sys,nom_fich)
    msg2 = (fstat,sys,nom_fich)

rp (done_open,FS,(nom_fich,cid)),(done_fstat,FS,(nom_fich,stat)):BAL
   (P,arg,pid,nom_fich,argv):BLK
if  cid ≠ NoSuchFile
```

```
    by (send,FS,msg):BAL, (P,arg,pid,cid,argv):BLK
        where
        msg = (read,sys,(cid,stat.length))

    rp (done_open,FS,(nom_fich,cid)),(done_fstat,FS,(nom_fich,stat)):BAL
       (P,arg,pid,nom_fich,argv):BLK
    if (cid = NoSuchFile) and (stat = NoSuchFile)
    by (P,arg',pid):SYS
        where
        arg' = cons(bal',tail(arg))
        bal' = head(arg) ∪ {msg}
        msg = (done_exec,sys,nom_fich,argv,NoSuchFile)

    rp (done_read,FS,(cid,data)):BAL, (P,arg,pid,cid,argv):BLK
    by (P',arg',pid):SYS
        where
        P' = F(data)
        arg' = argv

)
```

# B Programmes du Système Gestionnaire de Fichiers

## B.1 Le gestionnaire des requêtes aux fichiers

```
P_FS(⟨ BAL,CTX ⟩) = ⟨BAL',CTX'⟩
   where
   (BAL',CTX') = (gam BAL,CTX -> BAL,CTX
```

### B.1.1 Requête CREATE

```
   rp (create,exp,name):BAL
   by (send,NS,msg):BAL, (create,exp,name):CTX
       where
       msg = (createNS,FS,name)

   rp (done_createNS,NS,(name,fid)):BAL, (create,exp,name):CTX
   by (send,SS,msg1), (send,CS,msg2):BAL, (create,exp,name,fid):CTX
       where
       msg1 = (createSS,FS,fid)
       msg2 = (createCS,FS,fid)

   rp (done_createCS,CS,(fid,cid)):BAL, (done_createSS, SS, fid):BAL,
      (create,exp,name,fid):CTX
   by (send,exp,msg):BAL
       where
       msg = (done_create,FS,(name,cid))
```

### B.1.2 Requête OPEN

```
   rp (open,exp,name):BAL
   by (send,NS,msg):BAL, (open, exp,name):CTX
       where
       msg = (look_upNS,FS,name)

   rp (done_look_upNS,NS,(name,fid)):BAL, (open,exp,name):CTX
   by (send,CS,msg):BAL, (open,exp,name,fid):CTX
       where
       msg = (createCS,FS,fid)

   rp (done_look_upNS,NS,(name,NoSuchFile)):BAL, (open,exp,name):CTX
   by (send,exp,msg):BAL
       where
       msg = (done_open,FS,(name,NoSuchFile))

   rp (done_createCS,CS,(fid,cid)):BAL, (open,exp,name,fid):CTX
   by (send,exp,msg):BAL
       where
       msg = (done_open,FS,(name,cid))
```

### B.1.3 Requête CLOSE

```
rp (close,exp,cid):BAL
by (send,CS,msg):BAL, (close,exp,cid):CTX
    where
    msg = (closeCS,FS,cid)

rp (done_closeCS,CS,(cid,rep)):BAL
by (send,exp,msg):BAL
    where
    msg = (done_close,FS,(cid,rep))
```

### B.1.4 Requête READ

```
rp (read,exp,(cid,length)):BAL
by (send,CS,msg):BAL, (read,exp,cid,length):CTX
    where
    msg = (look_upCS,FS,cid)

rp (done_lookupCS,CS,(cid,pstn,fid,rep)):BAL,
   (read,exp,cid,length):MCTX
by msg:BAL, save:CTX
    where
    msg = if rep = Ok then {(send,SS,msg1)}
          else {(send,exp,msg2)}
    save = if rep = Ok then {(read,exp,cid,data,fid,pstn)}
           else ∅
    msg1 = (look_upSS,FS,fid)
    msg2 = (done_read,FS,(cid,NoSuchCid))

rp (done_look_upSS,(fid,file)):BAL,(read,exp,cid,length,fid,pstn):CTX
by (send,exp,msg1),(send,CS,msg2):BAL
    where
    msg1 = (done_read,FS,(cid,data))
    msg2 = (putCS,FS,(cid,pstn + long(data),fid))
    data = S(file,pstn,length)
```

### B.1.5 Requête WRITE

```
rp (write,exp,(cid,data)):BAL
by (send,CS,msg):BAL, (write,exp,cid,data):CTX
    where
    msg = (look_upCS,FS,cid)

rp (done_lookupCS,CS,(cid,pstn,fid,rep)):BAL,
   (write,exp,cid,data):CTX
by  msg:BAL, save:CTX
    where
    msg = if rep = Ok then {(send,SS,msg1)}
          else {(send,exp,msg2)}
```

```
        save = if rep = Ok then {(write,exp,cid,data,fid,pstn)}
               else ∅
        msg1 = (look_upSS,FS,fid)
        msg2 = (done_write,FS,(cid,NoSuchCid))

    rp (done_look_upSS,(fid,file)):BAL, (write,exp,cid,data,fid,pstn):CTX
    by (send,exp,msg1),(send,SS,msg2),(send,CS,msg3):BAL
        where
        msg1 = (done_write,FS,(cid,Ok))
        msg2 = (putSS,FS,(fid,file'))
        msg3 = (putCS,FS,(cid,pstn',fid))
        file' = modif(file,data',pstn)
        data' = zero append data
        zero = cons_l0(pstn - long(file))
        pstn' = pst + long(zero) + long(data)
        cons_l0 (c) = if c ≤ 0 then NIL else cons(0,cons_l0(c-1))

    )(BAL,CTX)
```

## B.2 Le système de nommage

```
P_NS(⟨BAL,MNS,MFID⟩) = ⟨BAL',MNS',MFID'⟩
    where
    (BAL',MNS',MFID') = (NS_OUT ∘ NS_IN)(BAL,MNS,MFID)
    NS_IN = (gam BAL,MNS,MFID -> BAL,MNS,MFID

    rp (createNS,exp,name):BAL, (name,fid):MNS
    by (send,exp,msg):BAL, (name,fid):MNS
        where
        msg = (done_createNS,NS,(name,fid))

    rp (look_upNS,exp,name):BAL, (name,fid):MNS
    by (send,exp,msg):BAL, (name,fid):MNS
        where
        msg = (done_look_upNS,NS(name,fid))

    rp (destroyNS,exp,name):BAL, (name,fid):MNS
    by (send,exp,msg):BAL, fid:MFID
        where
        msg = (done_destroyNS,NS,(name,ok))

    )
    NS_OUT = (gam BAL,MNS,MFID -> BAL,MNS,MFID

    rp (createNS,exp,name):BAL, fid:MFID
    by (send,exp,msg):BAL, (name,fid):MNS
        where
        msg = (done_createNS,NS,(name,fid))

    rp (look_upNS,exp,name):BAL
    by (send,exp,msg):BAL
        where
```

```
        msg = (done_look_upNS,NS,(name,NoSuchFile))

    rp (destroyNS,exp,name):BAL
    by (send,exp,msg):BAL
        where
        msg = (done_destroyNS,NS,(name,NoSuchCid))

    )
```

## B.3 Le système de canaux

```
P_CS(⟨BAL,MCS,MCID⟩) = ⟨BAL',MCS',MCID'⟩
    where
    (BAL', MCS', MCID') = (CS_OUT ∘ CS_IN) (BAL,MCS, MCID)
    CS_IN = ( gam BAL,MCS,MCID -> BAL,MCS,MCID

    rp (createCS,exp,fid):BAL, cid:MCID
    by (send,exp,msg):BAL, (cid,pstn,fid):MCS
        where
        msg = (done_createCS,CS,(fid,cid))
        pstn = 0

    rp (closeCS,exp,cid):BAL, (cid,pstn,fid):MCS
    by (send,exp,msg):BAL, cid:MCID
        where
        msg = (done_closeCS,CS,(cid,ok))

    rp (look_upCS,exp,cid):BAL, (cid,pstn,fid):MCS
    by (send,exp,msg):BAL, (cid,pstn,fid):MCS
        where
        msg = (done_look_upCS,CS,(cid,pst,fid,ok))

    rp (putCS,exp,(cid,fid,pstn)):BAL, (cid,fid,pst'):MCS
    by (cid,fid,pstn):MCS

    )
    CS_OUT = ( gam BAL,MCS,MCID -> BAL,MCS,MCID

    rp (closeCS,exp,cid):BAL
    by (send,exp,msg):BAL
        where
        msg = (done_closeCS,CS,(cid,NoSuchCid))

    rp (look_upCS,exp,cid):BAL
    by (send,exp,msg):BAL
        where
        msg = (done_look_upCS,CS,(cid,0,0,NoSuchCid))

    )
```

## B.4 Le système de stockage

```
P_SS(⟨BAL,MSS⟩) = ⟨BAL',MSS'⟩
    where
    (BAL', MSS') = (SS_OUT o SS_IN) (BAL,MSS)
    SS_IN = (gam BAL,MSS -> BAL,MSS

    rp (createSS,exp,fid):BAL (fid,file):MSS
    by (fid,NIL):MSS, (send, exp, msg):BAL
        where
        msg = (done_createSS, SS, fid)

    rp (look_upSS,exp,fid):BAL, (fid,file):MSS
    by (send,exp,msg):BAL, (fid,file):MSS
        where
        msg = (done_look_upSS,SS,(fid,file))

    rp (putSS,exp,(fid,file')):BAL, (fid,file):MSS
    by (fid,file'):MSS

    rp (destroySS,exp,fid):BAL, (fid,file):MSS
    by (send,exp,msg):BAL
        where
        msg = (done_destroySS,SS,(fid,ok))

    )
    SS_OUT = ( gam BAL,MSS -> BAL,MSS

    rp (createSS,exp,fid):BAL
    by (fid,NIL):MSS , (send, exp, msg):BAL
        where
        msg = (done_createSS, SS, fid)

    rp (look_upSS,exp,fid):BAL
    by (send,exp,msg):BAL
        where
        msg = (done_loo_upSS,SS,(fid,NoSuchFile))

    rp (destroySS,exp,fid):BAL
    by (send,exp,msg):BAL
        where
        msg = (done_destroySS,SS,(fid,NoSuchFile))

    )
```

# C Programme de l'interpréteur de commandes

```
P_shell (⟨BAL_SH,CTX_SH⟩) = ⟨BAL_SH',CTX_SH'⟩
   where
   (BAL_SH',CTX_SH') = sh(BAL_SH,CTX_SH)
   sh(BAL,CTX) = (gam BAL,CTX -> BAL,CTX

   rp (command,exp,parameters,bkg):BAL
   by (fork,pid_sh):BAL, (command,exp,parameters,bkg):CTX

   rp (done_fork,pid1),(pid_fils,pid2):BAL
      (command,exp,parameters,bkg), (pid_processus,pid_sh):CTX
   if (pid1 = 0)
   by (exec,command,arg'',pid2):BAL
       where
       arg'' = ⟨BAL_FILS,CTX_FILS⟩
       BAL_FILS = bag((command,parameters))
       CTX_FILS = bag((pid_père,pid_sh)) ∪ bag((pid_processus,pid2))

   rp (done_fork,pid1):BAL, (command,exp,parameters,bkg):CTX
   if (pid1 ≠ 0 and bkg)
   by (send,exp,msg):BAL
       where
       msg = (done_com_bkg,pid_sh)

   rp (done_fork,pid1), (done_com,pid1,etat):BAL
      (command,exp,parameters,bkg):CTX if (pid1 ≠ 0 and not bkg)
   by (send,exp,msg):BAL
       where
       msg = (done_com,pid_sh,(command,etat))

   )((BAL,CTX))
```

# D Programme de la commande *ls*

```
P_ls(⟨BAL_LS,CTX_LS⟩) = ⟨BAL_LS',CTX_LS'⟩
(BAL_LS',CTX_LS') = ( gam BAL,CTX -> BAL,CTX

   rp (ls,l_dir):BAL, (pid_processus,pid):CTX
   by (send,sys,msg):BAL, (ls,l_dir), (pid_processus,pid):CTX
       where
       msg = (get_arg,pid,pid_ns)

   rp (done_get_arg,sys,arg):BAL, (ls,l_dir), (pid_processus,pid),(pid_père,pid):CTX
   by (send,pid',msg1), (send,tty,msg2), (exit,pid,ok):BAL
       where
       msg1 = (done_com,pid,ok)
       msg2 = (print,files)
       files = unbag(FILES)
       FILES = search(NS,l_dir)
       NS = head(tail²(arg))
       search(N,l_d) = (gam L -> X -> Y
          rp l:L
          if tail(l) ≠ nil
          by tail(l):L, head(l):X

          rp dir:X
          by ⟨(dir,tri(⟨l_fich⟩))⟩:Y
             where
             ⟨l_fich⟩ = unbag(chercher(dir,N))
             chercher(d,N) = (gam N -> N
                rp (name,fid):N
                if (front(name) = d)
                by ⟨last(name)⟩:M

                rp l_fich,l_fich':M
                by l_fich append l_fich':M
             )N

          rp l_dir_fich, l_dir_fich' : Y
          by l_dir_fich append l_dir_fich' : Y
          )(bag(l_d))
```

# E Programme de la commande *ls**

```
P_ls*(⟨BAL_LS,CTX_LS⟩) = ⟨BAL_LS',CTX_LS'⟩
   where
   (BAL_LS',CTX_LS') = ( gam BAL,CTX -> BAL,CTX

   rp (ls,l_dir):BAL, (pid_pr,pid):CTX
   by (send,sys,msg):BAL, (ls,l_dir), (pid_pr,pid):CTX
       where
       msg = (get_arg,pid,pid_ns)

   rp (done_get_arg,sys,arg):BAL, (ls,l_dir), (pid_pr,pid):CTX
   by (send,exp,msg1), (send,tty,msg2), (exit,pid,ok):BAL
       where
       msg1 = (done_com,pid,ok)
       msg2 = (print,files)
       files = tri(unbag(FILES))
       FILES = search (NS,dir,ter)
       search(N,d,t) = (gam N -> N'
          rp (name,fid):N
          if (front(name) = d) and (end(last(name),t)
          by ⟨last(name)⟩:N'

          rp ⟨name⟩, ⟨name'⟩: N'
          by ⟨name⟩ append ⟨name'⟩ : N'
       )
```

# F Programme de l'utilitaire *make*

```
Pmake = (gam(BAL,CTX,M,M',N,R) -> (BAL,CTX,M,M',N,R)

   rp (make,nom_prog):BAL
   by (send,FS,msg1),(send,FS,msg):BAL, (make,nom_prog):CTX
       where
       msg1 = (open,id_mk,Makefile)
       msg2 = (fstat,id_mk,Makefile)

   rp (done_open,FS,(Makefile,cid)), (done_fsta,FS,(Makefile,etat):BAL
   by (send,FS,msg):BAL
       where
       msg = (read,id_mk,(cid,etat.length))

   rp (done_read,FS,(cid,liste_octets)):BAL, (make,nom_prog):CTX
   by (elimdoubles o triplets_out o triplets_in o analyse)
      (liste_octets,nom_prog):N
       where

       (*(1)
       analyse est une fonction, qui étant donnée une liste d'octets représentant un
       fichier makefile, rend la paire (FMAKEFILE,nom_prog) oàu FMAKEFILE est un
       multiensemble de triplets de la forme (c,D,P), un triplet particulier
       correspondant à une règle de makefile (c est le nom d'un fichier, D est
       l'ensemble des fichiers dont dépend c, P est le procédé de mise à jour).*)

       (*2*)
       (triplets_in) (FMAKEFILE,nom_prog) =
       (gam FMAKEFILE -> FMAKEFILE
          rp (prog',D',P'):FMAKEFILE, (make, nom_prog):CTX
          if prog' = nom_prog
          by (prog,D',P'):N, D':Md

          rp d:Md, (d',D',P'):FMAKEFILE
          if d = d'
          by (d,D',P'):N, D':Md
       ) FMAKEFILE

       (* Ce programme rend un multiensemble FMAKEFILE' contenant tous les triplets
       de FMAKEFILE associés aux fichiers intervenant dans la mise à jour du fichier
       nom_prog. *)

       (*3*)
       (triplets_out) (FMAKEFILE') =
       ( gam Md -> N
          rp d:Md
          by (d,∅,skip):N
       )FMAKEFILE'

       (* Ce programme rajoute dans N, pour chaque fichier "source" et impliqué, un
       triplet (d,{},skip).*)

       (*4*)
```

```
    (elimdoubles) N =
    (gam N -> N
       rp (c,D,P):N, (c',D',P'):N
       if c = c' and D = D' and P = P'
       by (c,D,P):N
    )N

    (* Ce programme élimine de N les occurrences multiples de triplets. *)

(* Le multiensemble N une fois construit, les réactions suivantes mettent en
œuvre le reste du programme Pmake. *)

rp (c,D,P):N
by (send,FS,msg):BAL, (c,D,P,D,false):M'
    where
    msg = (fstat,id_mk,c)

rp (done_fstat,FS,(name,etat)):BAL, (c,D,P,d,b):M'
if (name = c)
by (c,D,P,d,b,etat.date):M

rp (c,D,P,d,b,date):M
if (d = ∅) and (not b)
by (c,date):R

rp (c,D,P,d,b,date):M, (r,date'):R
if (r in d) and (date' ≤ date)
by (c,D,P,d-c,b,date):M, (r,date'):R

rp (c,D,P,d,b,date):M, (r,date'):R
if (r in d) and (date' > date)
by (c,D,P,d-c,true,date):M, (r,date'):R

rp (c,D,P,d,b,date):M
if (d = ∅) and (not b)
by (c,D,P,b,date):CTX, (fork,id_mk):BAL

rp (done_fork,pid),(done_P,x,c):BAL
if (pid ≠ 0)
by (send,FS,msg):BAL
    where
    msg = (fstat,id_mk,c)

rp (done_fstat,FS,(name,etat)):BAL, (c,D,P,b,date):CTX
if (name = c)
by (c,etat.date):R

rp (done_fork,pid),(pid_fils,pid'):BAL, (c,D,P,b,date):CTX
if (pid = 0)
by (exec,P,arg,pid'):BAL
    where
    arg = ⟨BAL'⟩
    BAL' = D ∪ (pid_père,id_mk),(pid_fils,pid')
```

## LISTE DES DERNIERES PUBLICATIONS INTERNES IRISA

PI 586 OPAC : A COST-EFFECTIVE FOATING-POINT COPROCESSOR
André SEZNEC, Karl COURTEL
Mai 1991, 26 Pages.

PI 587 ON FAILURE DETECTION AND IDENTIFICATION : AN OPTIMUM ROBUST MIN-MAX APPROACH
Elias WAHNON, Albert BENVENISTE
Mai 1991, 24 Pages.

PI 588 BOUNDED-MEMORY ALGORITHMS FOR VERIFICATION ON THE FLY
Claude JARD, Thierry JERON
Mai 1991, 14 pages.

PI 589 UNE APPROCHE MULTIECHELLE A L'ANALYSE D'IMAGES PAR CHAMPS MARKOVIENS
Patrick PEREZ, Fabrice HEITZ
Juin 1991, 32 pages.

PI 590 THE IDEMPOTENT SOLUTIONS OF THE SEMI-UNIFICATION PROBLEM
Pascal BRISSET, Olivier RIDOUX
Juin 1991, 16 pages.

PI 591 AVARE UN PROGRAMME DE CALCUL DES ASSOCIATIONS ENTRE VARIABLES RELATIONNELLES
Mohamed OUALI ALLAH
Juin 1991, 32 pages.

PI 592 SCHEDULING IN DISTRIBUTED SYSTEMS : SURVEY AND QUESTIONS
Yasmina BELHAMISSI, Maurice JEGADO
Juin 1991, 36 pages.

PI 593 APPLICATION OF BELLEN'S PARALLEL METHOD TO ODE's WITH DISSIPATIVE RIGHT-HAND SIDE
Philippe CHARTIER
Juin 1991, 24 pages.

PI 594 PROGRAMMATION D'UN NOYAU UNIX EN GAMMA
Pascale LE CERTEN, Hector RUIZ BARRADAS
Juillet 1991, 48 pages.



Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique.

ISSN 0249 - 6399